Predicate/Transition Net Simulation based on Concurrent Prolog(Software Science and Engineering)

AUTHOR(S):

Mizuba, Hideyo; Harath, Jayantha; Ueda, Kenji; Saito, Nobuo

数理解析研究所講究録
第547巻 1985年 23-34

Predicate/Transition Net Simulation based on Concurrent Prolog

Hideyo Mizuba†, Jayantha Herath†, Kenji Ueda‡, Nobuo Saito†

†Department of Mathematics
Keio University
Yokohama, JAPAN

‡Software Research Center
Technology Division
Ricoh Company, Ltd.
Tokyo, JAPAN

ABSTRACT

This paper first describes the predicate transition nets and Concurrent Prolog programming language briefly. An application of Concurrent Prolog is then introduced as simulation of predicate transition nets. Then the merits and demerits of concurrent prolog are discussed. Finally simulation results are introduced and discussed.

1. Introduction

The Predicate/transition net is an extension of Petri net introduced by [1], and it has the ability to model various general properties of concurrent, distributed and asynchronous processes[2]. Each token in a predicate node of a predicate/transition net has its type or label. A transition can be fired if all its input predicate tokens are available. Arcs from predicate nodes to a transition have labels with polynomial equations of variables appearing in the transition formula.

Concurrent or distributed processing system specifications can be described clearly by using predicate/transition nets. These descriptions can be used for rapid prototyping of the target concurrent processing system, and for system design verification. Therefore it is useful to construct a simulation system to simulate a given predicate/transition net. This paper describes a simulation method for predicate/transition nets.

Concurrent Prolog, a logic programming language with concurrency, has been recently introduced by Shapiro [3]. This paper first briefly describes the predicate/transition net model and then the special characteristics of Concurrent Prolog. The property identified between concurrent prolog and predicate/transition net is then described and used to predicate/transition net simulation. The advantages and disadvantages of using Concurrent Prolog as a simulation language for simulating the predicate transition net model are discussed. A comparison of concurrent prolog to other languages in predicate/transition net simulating is also given.

2. Predicate/Transition Net

Here we briefly discuss the basic properties of place/transition nets (Petri nets) [6][7], condition/event nets (C/E nets)[8] and predicate/transition nets (Pr/T nets)[8].

2.1. Place/Transition nets

The Petri net, a parallel flow graph, originated with [1]. It is a simple, natural and powerful method of describing the information flow control of asynchronous parallel or distributed systems. This makes Petri net an excellent working tool for such systems. A Petri net graph models the static properties of a system and the dynamic properties result by executing the graph. It can be considered as a collection of transitions, places and arcs. Arcs connect the transitions and places, and are used as paths to transfer information between places and transitions. Places can be regarded as the temporary storages to store the tokens which are processed by transitions.

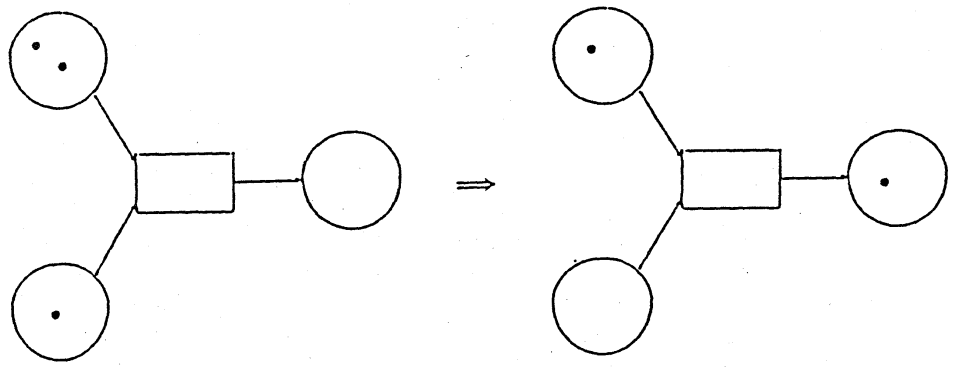


Fig. 2.1 Simple Petri Net

Fig. 2.1 illustrates a simple Petri net. Here a box represents the transition, a circle represents the places, and dots within places are called tokens. When all the input tokens of a transition are available, the transition is fired. Firable transitions are selected non-deterministically and are fired asynchronously. The Petri net execution is controlled by token distribution (called marking). A fired transition generates new output tokens and transfers them to the output places.

Petri nets are powered by handling constraints, exclusive OR transitions, switches, inhibitor arcs and time. Petri nets have been extensively used because of their capability of clearly describing concurrency, conflicts and synchronization of processes. Petri net is used to model various kinds of asynchronous parallel or distributed system models such as concurrent computing systems. Original Petri nets were used to represent the logical behaviour of systems and were often used to model von Neumann computing systems. One of the major recent applications of Petri net concept is modeling new generation computing systems such as dataflow and demandflow computers.

2.2. Condition/Event Nets

The basic and most natural interpretation of Petri net is the condition/event system model. There are many systems (called discrete event systems) where events occur in discrete periods of time, and the system status varies according to the event occurences. The places in a Petri net represent the conditions, while the transitions represent the events of the C/E net. A firing of a transition corresponds to the event occurence, and the change of its marking corresponds to the system status transition. This is very naive interpretation, but it is possible to model various discrete systems in the condition/event net.

2.3. Predicate/Transition Nets

The predicate/transition net is an extension of the Petri net and is used to model concurrent or distributed systems. The role played by integers in ordinary Petri nets extends to integer polynomials in the predicate/transition net. Predicate/transition nets are based on ordinary Petri net schemes. Places in the Petri net are identified as predicates. A transition has a specified formula to determine its firing. All arcs are labled with polynomials of variables in input predicates. This increases the descriptive power of predicate/transition nets. A transition is fired when all respective tokens of input arcs are available. This is the only condition necessary to satisfy the execution of a transition formula. The tokens generated by the transition execution are transferred to the related arcs in the output.

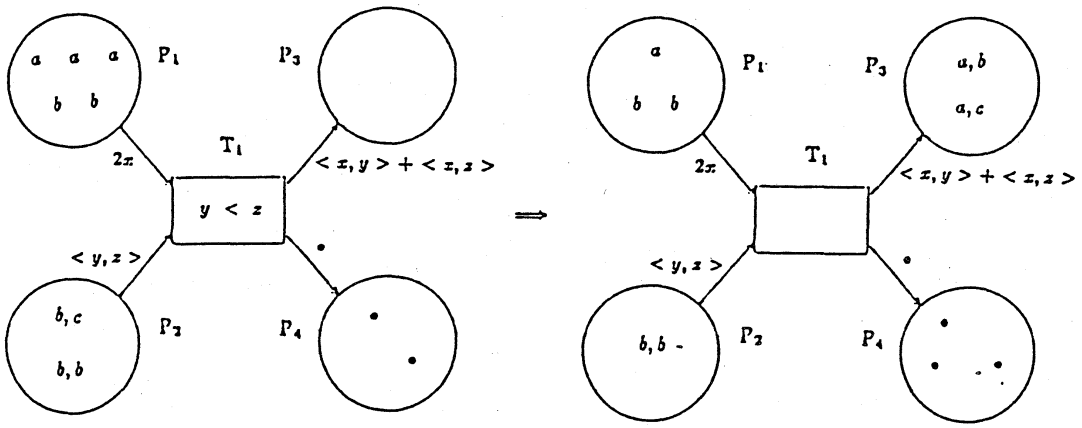


Fig. 2.2 Example – Transition Rule

In predicate/transition nets, a transition is a petri net transition schemes, a predicate is a petri net place schemes and arcs are labelled. This reduces the net size of condition/event nets and Petri nets. Predicate/transition nets are generalized Petri nets powered by both first order predicate logic and linear algebra. In Fig. 2.2, P_1, P_2, P_3 and P_4 are predicates. 'a' tokens and 'b' tokens have P_1 property and 'b, c' token and 'b, b' token have P_2 property. P_3 predicate has no predicates initially. P_4 is a no argument predicate. P_1 and P_2 are variable properties and P_3 and P_4 are variable relation. T_1 is 'Y<Z' transition. Input and output arcs are labled as shown in Fig. 2.2.

3. Concurrent Prolog

In this section the concepts of logic programming, Prolog and Concurrent Prolog are discussed briefly.

3.1. Logic Programming [9]

Lisp and Prolog are AI languages. Lisp is a functional or an object oriented language. Object oriented languages make little distinction between data and program.

Logic programming is suitable for dealing with large data bases. A time independent fact collection can be used to solve problems related to creating and/or recognizing patterns, that is, to solve problems whose solution is a deductive logic function.

A definite clause in logic programming is written as

$$A :- B_1,\ B_2,\ B_3,\ ...,\ B_n \quad n \geq 0 \qquad (1)$$

Two ways of reading (1), namely, declarative reading and procedural reading are described below. Another way of reading of (1), namely, behavioural reading is described in section 3.3.

1) Declarative reading

A is true if B_1 and B_2 and ... and B_n are true.

2) Procedural reading

To execute procedure call A , perform procedure calls B_1 , B_2 ..., and B_n. Or to solve problem A, solve sub problems B_1 , B_2 ..., and B_n.

3.2. Sequential Prolog [10]

Prolog is a typical logic programming language, although it is still experimental. Prolog's roots in predicate logic give it a built-in relational data base. Programming in prolog can be viewed as programming by assertion and query. The prolog programmer need not aware of the distinction between programming and querying. The information stored in the internal relational database of prolog can be retrieved by using powerful relational queries. Prolog's implicit search strategy is good for symbolic processing. The name of the prolog relationship is called predicate. Sequential prolog is designed to run efficiently in von Neumann computing environment.

Lisp is a more fundamental language and gives better control over processes so the program won't waste time on useless searches. It is not clear whether the prolog can give enough control and the efficiency needed for nontrivial problems. Prolog is not an efficient implementation language for many sequential algorithms.

3.3. Concurrent Prolog

Concurrent Prolog proposed by E. Shapiro[3] is a prolog variant and is closely related to relational languages[11]. Concurrent Prolog is an efficient language for implementing many parallel algorithms, and allows logic programs to be executed concurrently[4,5].

Object oriented programming concepts and techniques are realized naturally in Concurrent Prolog. Basic operations of object oriented programming such as creating objects, sending and receiving messages, modifying object state, and forming class- super class hierarchies are supported in Concurrent Prolog. Lisp functions are easily implemented in Concurrent Prolog.

Read only variables and variables in a process, are introduced to support process synchronization.

Message passing in Concurrent Prolog adds the power of destructive assignment, which is not available in Sequential Prolog. Message passing and structure copying capabilities make the language a powerful concurrent programming language. Partially determined messages are very useful and powerful.

Guarded clauses with read only variables give wide range of indeterminate process behaviour.

A set of guarded clauses constructs a Concurrent Prolog program. All guards and all elements of a conjunctive goal are executed concurrently. A guarded clause of concurrent prolog is as follows:

$$H \ :- \ G_1, \ G_2, \ ..., \ G_m \mid B_1, \ B_2, \ ..., \ B_n \quad m, \ n \geq 0. \tag{2}$$

where G's and B's are unit goals. B's are executed when G's succeeds.
H - head
G - guard
B - body
lower case - constants
upper case - variables
$[X|Y]$ - list with head X and tail Y.

Concurrency supporters of Concurrent Prolog

| - commit operator — concurrency supporter for guards.
$X?$ - read only variables — concurrency supporter for process communication and synchronization.

In Concurrent Prolog, all conjunctive goals are solved simultaneously.

Table 3.1 shows the traversing orders of logic, sequential prolog and concurrent prolog interpreters.

Logic interpreter	**Sequential Prolog interpreter**	**Concurrent Prolog interpreter**
Correct non-deterministic choices at OR nodes and traverse AND nodes arbitrarily in a AND OR tree.	depth first, left to right traversing order conjunctive goals reduced from left to right. If there are many alternate ways to reduce a unit goal, they are fired one by one using backtracking.	All guards and all elements of a conjunctive goals are executed simultaneously.

Table 3.1 Traversing Orders

1) Behavioural reading

This is the logic program reading method in Concurrent Prolog. The logic program (1) is read as, a process A can be replaced by a process system containing B_1 and B_2 and .... B_n. The process replaced by empty system causes the process to terminate.

The Concurrent Prolog clause (2) is read as, the process H can be replaced by a process system containing G's and B's.

4. Pr/T net simulation using Concurrent Prolog

Parallel and/or distributed system modeled using Pr/T nets have been simulated so far using either sequential or concurrent programming languages. In this section the relationship between Pr/T net and Concurrent Prolog, the simulation method used to simulate Pr/T net using Concurrent Prolog and a simulation example will be described.

4.1. Relation between Pr/T net and Concurrent Prolog

The parallel and/or distributed systems can be modeled using the net theory which represents the logical behaviour of the system elements. The extension of Petri net, Pr/T net is a generalized Petri net powered by predicate logic and linear algebra. Concurrent Prolog is a logic programming language with the ability to express concurrent logical system actions and execute them concurrently. The behaviour of the Pr/T net model is used as the concurrent system.

There is an intuitive similarity in Pr/T and Prolog. Fig 2.2 illustrates this fact. The transition with first order predicate in Pr/T is directly described as a goal in a clause. It is firable if the unifications for input arcs can satisfy the goal(the predicate). The places in Pr/T net can be described as facts in Prolog database, and the state transition(marking transition) can be caught up by changing the database. Since there are several numbers of transitions in Pr/T net, it is better to use Concurrent Prolog because it has a scheduler for firable transitions (executable prolog goals). It is necessary to non-deterministically select one of them, and Concurrent Prolog scheduler does it. Scheduling algorithm can be programmed in Concurrent Prolog, and this flexibility will be very useful for Pr/T net simulation. Table 4.1 shows the relationships between Pr/T net and Concurrent Prolog. The simulation example in this section illustrates the power of this idea.

Pr/T net	Concurrent Prolog
n-tuple	tuple
predicate	predicate
transition rule	rule
marking	fact
execution	goal(solve)

Table 4.1 Similarity between Objects in Pr/T net and Concurrent Prolog

4.2. Simulation Method

Concurrent Prolog description of the Pr/T net model and how the simulation is performed are described in this section. To simulate Pr/T net using Concurrent Prolog, it is necessary to represent each Pr/T net object by Concurrent Prolog primitive. For example, a predicate of a Pr/T net can be represent as a predicate in the internal database of Concurrent Prolog.

The simulation method is given as follows:

Step 1 Label each predicate and each transition in the directed net.
Step 2 Define the followings using Concurrent Prolog Predicates with the label given in step 1 as the predicate name. 1. The n-tuple variable of each transition. 2. A quantifier-free logical formula built from equality, operators and predicates. 3. Variables occurring at surrounding arcs.
Step 3 Add initial tokens to the prolog internal database.
Step 4 Interpret the predicates asserted in internal database in step 2 concurrently using concurrent prolog.

Several new database manipulation tools are used as Concurrent Prolog predicates to simulate Pr/T nets. The new predicates are named as **enable**, **remove** and **deposit**. The functions of these new predicates are as follows:

enable(*tuples*, *predicate*) — check whether *tuples* are taken out from predicate specified by *predicate*.

remove(*tuples*, *predicate*) — take out *tuples* from predicate specified by *predicate*.

deposit(*tuples*, *predicate*) — put *tuples* into predicate specified by *predicate*.

Note: The internal prolog database will be changed after the evaluation of these predicates. Detailed simulator program is in Appendix A.

4.3. Simulation

The simulation programs describing Pr/T nets are executed using the Concurrent Prolog interpreter which runs under the C-Prolog interpreter. The example shown below gives the Pr/T net model used for simulation and its corresponding Concurrent Prolog program.

```
p1([.(a),.(a),.(b),.(b),.(b)]).
p2([.(b,c),.(b,b)]).
p3([]).
p4([0,0]).

consystem(f1).

t1 :- f1, t1.

f1 :-
     enable([.(X),.(X)], p1),
     enable([.(Y,Z)], p2),
     !, aless(Y, Z)
     ->
       write(t1), nl,
       remove([.(X),.(X)], p1),
       remove([.(Y,Z)], p2),
       deposit([.(X,Y),.(X,Z)], p3),
       deposit([0], p4)
     ;
       true.
f1 :- true


?- execute(t1).
```

Program 4.1 Simulation program of Fig 2.2

1) Concurrent prolog can execute the Pr/T net concurrently by using its concurrent property.
2) Pr/T nets and Concurrent Prolog are based on first order logic. This makes the mapping simple.
3) The unification is simple because the tokens in Pr/T net correspond to the tuples in Prolog.

Example 1.

The simulation program of Pr/T net, shown in Fig. 2.2, is described using Concurrent Prolog in program 4.1.

0 denotes the zero tuple of Pr/T nets. Predicate *consystem* is for system predicate dealings, i.e., in program 4.1 the predicate `f1` represents an indivisible part in Concurrent Prolog. Appendix B gives another Pr/T net example and its Concurrent Prolog program used in this simulation study.

5. Discussion

In general, Pr/T-nets can be mapped naturally with Concurrent Prolog due to the combination of genuine sequential prolog properties and Concurrent Prolog properties. The advantages and difficulties of using concurrent prolog in this simulation study are discussed below.

5.1. Advantages of Using Concurrent Prolog

The advantages of simulating concurrent distributed systems such as Pr/T net using Concurrent Prolog are:

5.2. Difficulties of using Concurrent Prolog

As we have described in section 4, inter-process communication mechanism in concurrent prolog is used for marking tokens among predicates in the simulation. To use inter process communication, it is necessary to denote both transitions and predicates as processes in Concurrent Prolog. Inter process communication in Concurrent Prolog only communicates between certain related processes. It is difficult to describe a general cyclic graph using the inter process communication facilities of concurrent prolog.

5.3. Future Research

It is necessary to research further to identify the power and/or weaknesses of Concurrent Prolog language. Our future research plans will be briefly described in the following sections.

1) Time dependency

Existing time dependency handling languages are not suitable for handling Artificial Intelligence applications. Formal mathematical logic has no consistent, rational means of handling the concept of time. It is necessary to research designing time dependency handling logic programming languages for Artificial Intelligence applications. We intend to continue our research on finding ways of handling time dependencies in Concurrent Prolog and also to add the time constraint to predicate transition nets. This will help to simulate timed predicate transition nets using timed Concurrent Prolog language.

2) Comparison Concurrent Prolog – other Languages

Research has been conducted on comparison of Concurrent Prolog with other programming languages such as Ada, C and Concurrent Pascal for the concurrent and/or distributed system applications.

6. Conclusion

In this paper, Petri nets, C/E nets and Pr/T nets and the properties of logic programming were briefly discused. Two logic programming methods in use are Sequential Prolog and Concurrent Prolog. The properties of Concurrent Prolog logic programming language were also described.

The importance of simulating concurrent and/or distributed systems with concurrent logic programming language was described and a Pr/T net model was simulated using Concurrent Prolog. The simulation results, advantages and disadvantages of using Concurrent Prolog as a simulation language for concurrent system simulation were discussed. It is observed that the Concurrent Prolog inter process communication mechanism makes difficulties in simulation. Concurrent prolog can be used as a Petri net language. Future research topics, time dependence of AI languages and comparison of Concurrent Prolog with other languages, were discussed to over come the shortcomings of Concurrent Prolog. In many ways Concurrent Prolog is a suitable programming language for simulating concurrent and/or distributed systems.

References

[1] Hartmann J. Genrich and Kurt Lautenbach, "The Analysis of Distributed Systems by Means of Predicate/Transition Nets," Proc. of Semantics of Concurrent Computation, *Lecture Notes in Computer Science*, No. 70, Springer-Verlag, 1979.

[2] B. Gerard and T. Richard, "Modeling and Proofs of a Data Transfer Protocol by Predicate/Transition Nets," Application and Theory of Petri Nets, *Informatik Fachbericthe*, No. 52, Springer-Verlag, 1982.

[3] E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," TR-003, ICOT, Febuary 1983.

[4] E. Y. Shapiro, "System Programming in Concurrent Prolog," TR-034, ICOT, November 1983.

[5] David Gelernter, "A Note on Systems Programming in Concurrent Prolog," *1984 International Symposium on Logic Programming*, IEEE Computer Society Press, 76-82.

[6] C. Petri, "Fundamentals of a Theory of Asynchronous Information Flow," Information Processing 62, Proceedings of the 1962 IFIP Congress, North-Holland, 386-390, August 1962.

[7] James L. Peterson, Petri net Theory and Modeling of Systems, Prentice Hall, Inc., 1981.

[8] H.J. Genrich, K. Lautenbach, P.S. Thiagarajan, "Net Theory And Application," *Lecture Notes in Computer Science*, No.84 Springer-Verlag, 21-159, 1979.

[9] Robert Kowalski, "Predicate Logic as Programming Language," *Information Processing 74*, North-Holland, 569-574, 1974.

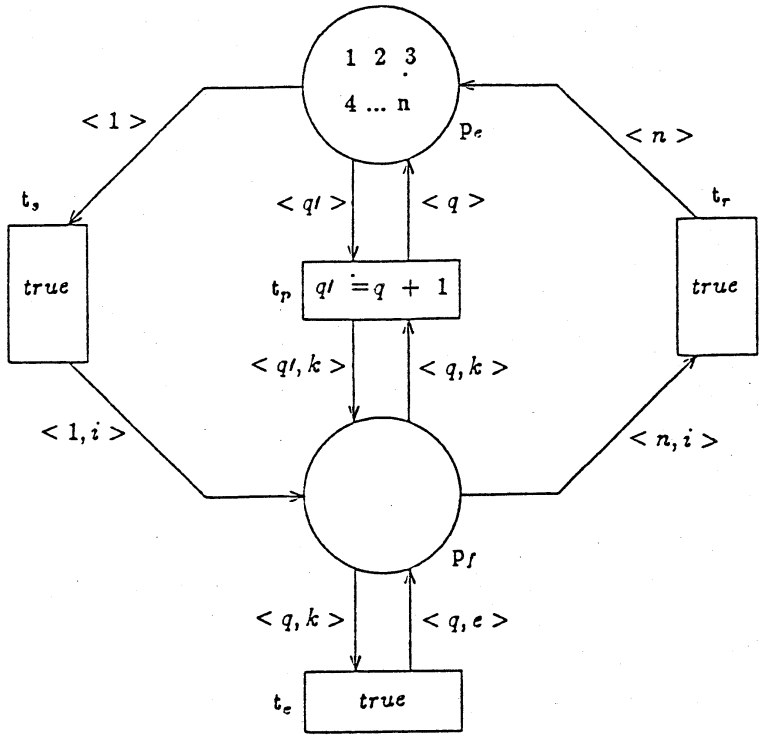
[10] W.F. Clocksin, C.S. Mellish, Programming in Prolog, Springer-Verlag, 1981.

[11] K.L. Kowalski and S.Gregory, "A Relational Language for Parallel Programming," *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, ACM, October, 1981.

Appendix A. Support Program for Simulation

```
enable(Tuples, Func) :-
     Pred1 =.. [Func, Tokens1],
     call(Pred1),
     enable2(Tuples, Tokens1),
     retract(Pred1),
     shuffle(Tokens1, Tokens2),
     Pred2 =.. [Func, Tokens2],
     assert(Pred2).

enable2([], Tokens1).
enable2([Tuple|Tuples], Tokens1) :-
     match(Tuple, Tokens1, Tokens2),
     enable2(Tuples, Tokens2).

shuffle(Tokens1, Tokens2)
     :- % for non-deterministic execution

match(X, [], []) :- !, fail.
match(X, [X|T], T).
match(X, [Y|T], R) :- match(X, T, R).

remove(Tuples, Func) :-
     Pred1 =.. [Func, Tokens1],
     call(Pred1),
     remove2(Tuples, Tokens1, Tokens2),
     retract(Pred1),
     Pred2 =.. [Func, Tokens2],
     assert(Pred2),
     write(Pred2), nl.

remove2([], Tokens, Tokens).
remove2([Tuple|Tuples], Tokens1, Tokens2) :-
     subtract(Tuple, Tokens1, Tokens3), !,
     remove2(Tuples, Tokens3, Tokens2).

subtract(X, [], []) :- % This goal must not be reached as usual.
     write('Fatal error'), !, fail.
subtract(X, [X|Y], Y).
subtract(X, [Y|Z], [Y|W]) :- subtract(X, Z, W).

deposit(Tuples, Func) :-
     Pred1 =.. [Func, Tokens1],
     call(Pred1),
     append(Tokens1, Tuples, Tokens2),
     retract(Pred1),
     Pred2 =.. [Func, Tokens2],
     assert(Pred2),
     write(Pred2), nl.
```

Appendix B. An Example for Simulation




```
pe([.(1),.(2),.(3),.(4),.(5)]).
pf([]).

consystem(fs).
consystem(fe).
consystem(fr).
consystem(fp).

ts :- fs, ts.
te :- fe, te.
tr :- fr, tr.
tp :- fp, tp.

fs :-
     enable([.(X)], pe),
     !, X = 1 ->
     write(ts), nl,
     remove([.(X)], pe),
     deposit([.(1,i)],pf);
     true.
fs :- true.
```

34

```
fe :-
     enable([.(Q,K)], pf),
     !, true ->
     write(te), nl,
     remove([.(Q,K)], pf),
     deposit([.(Q,e)], pf);
     true.
fe :- true.

fp :-
     enable([.(R)],pe),
     enable([.(Q,K)],pf),
     !, S is Q + 1, R = S ->
     write(tp), nl,
     remove([.(R)],pe),
     remove([.(Q,K)],pf),
     deposit([.(R,K)],pf),
     deposit([.(Q)],pe);
     true.
fp :- true.

fr :-
     enable([.(N,J)], pf),
     !, N = 5 ->
     write(tr), nl,
     remove([.(N,J)], pf),
     deposit([.(N)], pe);
     true.
fr :- true.
```